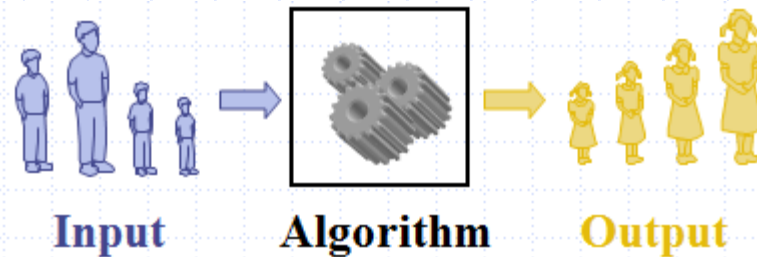


## Lesson 2: Introduction to Analysis of Algorithms: *Discovering the Laws Governing Nature's Computation*



### **Wholeness of the Lesson**

An algorithm is a procedure for performing a computation or deriving an output from a given set of inputs according to a specified rule. By representing algorithms in a neutral language, it is possible to determine, in mathematical terms, the efficiency of an algorithm and whether one algorithm typically performs better than another. Efficiency of computation is the hallmark of Nature's self-referral performance. Contact with the home of all the laws of nature at the source of thought results in action that is maximally efficient and less prone to error.

# Natural Things to Ask

- ◆ An algorithm is a procedure or sequence of steps for computing outputs from given inputs.
  - How can we determine whether an algorithm is *efficient* ?
  - Given two algorithms that achieve the same goal, how can we decide which one is better? (E.g. sorting) Can our analysis be independent of a particular operating system or implementation in a language?
  - How can we express the steps of an algorithm without depending on a particular implementation?

# A Framework For Analysis of Algorithms

We will specify:

- ◆ A simple neutral language for describing algorithms
- ◆ A simple, general computational model in which algorithms execute
- ◆ Procedures for measuring running time
- ◆ A classification system that will allow us to categorize algorithms (a precise way of saying “fast”, “slow”, “medium”, etc)

# Overview of the Lesson

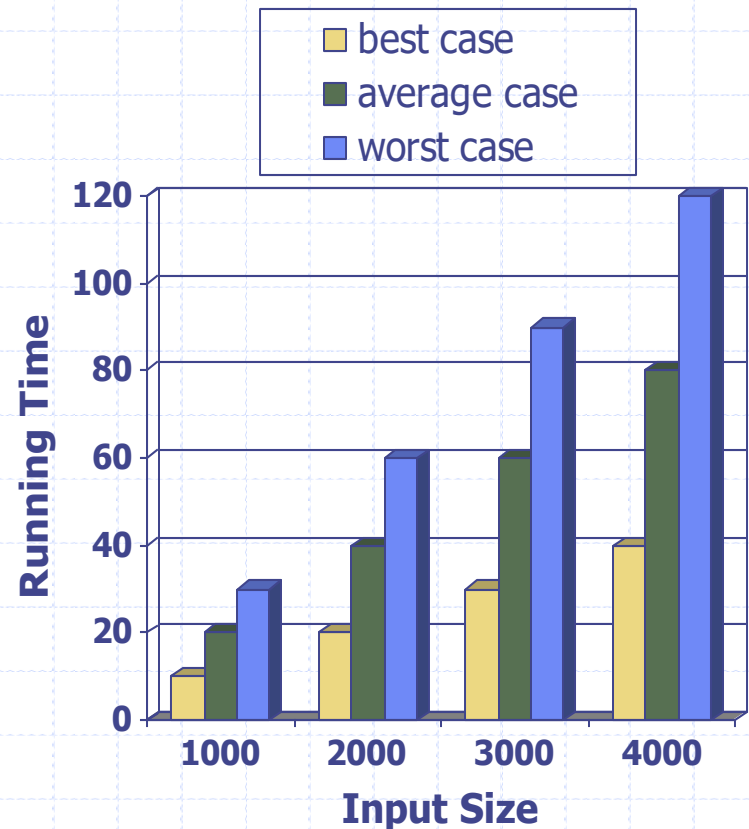
- Issues in Determining Running Time
- Pseudo-code to Describe Algorithms
- RAM Model and Counting Primitive Operations
- Asymptotic Algorithm Analysis
- Running Time of Recursive Algorithms
  - Guessing Method
  - Counting Self-Calls
  - The Master Formula

# Overview of the Lesson

- Issues in Determining Running Time
- Pseudo-code to Describe Algorithms
- RAM Model and Counting Primitive Operations
- Asymptotic Algorithm Analysis
- Running Time of Recursive Algorithms
  - Guessing Method
  - Counting Self-Calls
  - The Master Formula

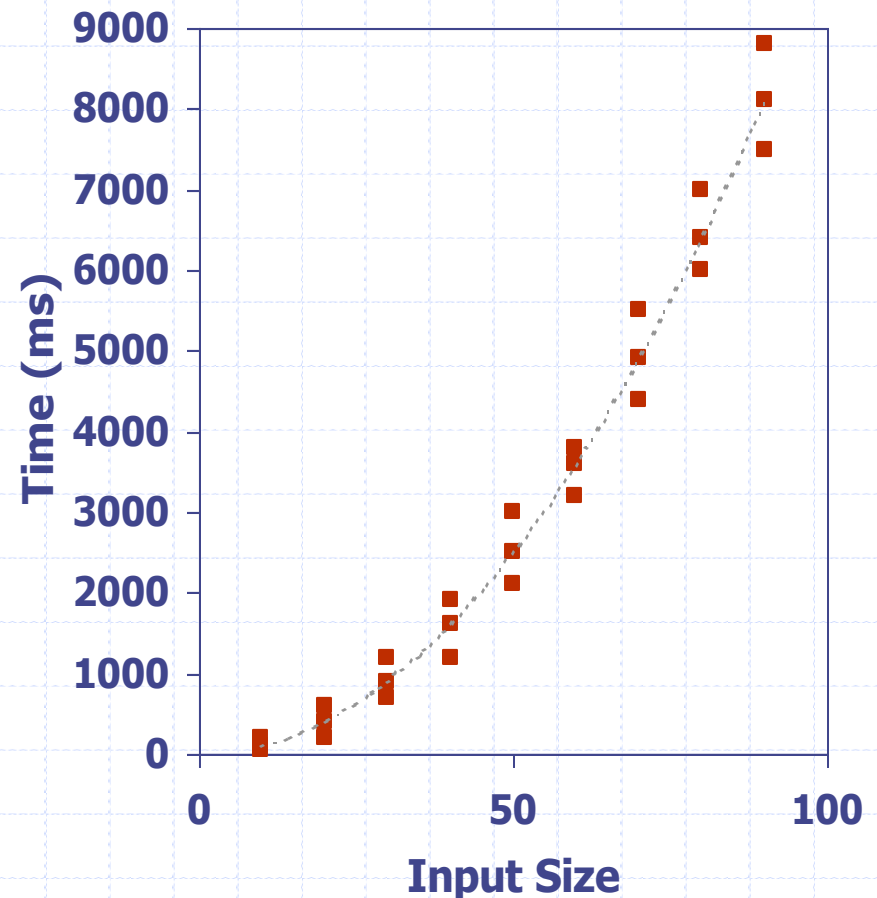
# Issues in Determining Running Time

- ◆ The running time of an algorithm typically grows with the input size.
- ◆ Average case time is often difficult to determine.
- ◆ Often, we focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



# Direct Measurement

- ◆ Write a program implementing the algorithm
- ◆ Run the program with inputs of varying size
- ◆ Use a method like `System.currentTimeMillis()` to get an accurate measure of the actual running time
- ◆ Optionally, plot the results



# Limitations of Direct Measurement

- ◆ It is necessary to implement the algorithm, which may be difficult
- ◆ Results may not be indicative of the running time on other inputs not included in the experiment.
- ◆ In order to compare two algorithms, the same hardware and software environments must be used





# Theoretical Analysis



- ◆ Uses a high-level description of the algorithm instead of an implementation
- ◆ Characterizes running time as a function of the input size,  $n$ .
- ◆ Takes into account all possible inputs
- ◆ Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

# Overview of the Lesson

- Issues in Determining Running Time
- Pseudo-code to Describe Algorithms
- RAM Model and Counting Primitive Operations
- Asymptotic Algorithm Analysis
- Running Time of Recursive Algorithms
  - Guessing Method
  - Counting Self-Calls
  - The Master Formula

# Pseudo-code to Describe Algorithms

- ◆ High-level description of an algorithm
- ◆ More structured than English prose
- ◆ Less detailed than a program
- ◆ Preferred notation for describing algorithms

*Example:* find max element of an array

```
Algorithm arrayMax(A, n)  
  Input array A of n integers  
  Output maximum element of A  
  
  currentMax  $\leftarrow A[0]$   
  for i  $\leftarrow 1$  to n - 1 do  
    if A[i] > currentMax then  
      currentMax  $\leftarrow A[i]$   
  return currentMax
```

# Pseudocode Syntax



## ◆ Control flow

- **if ... then ... [else ...]**
- **while ... do ...**
- **repeat ... until ...**
- **for ... do ...**
- Indentation replaces braces

## ◆ Method declaration

**Algorithm** *method* (*arg* [, *arg*...])

**Input** ...

**Output** ...

## ◆ Method call

*var.method* (*arg* [, *arg*...])

## ◆ Return value

**return** *expression*

## ◆ Expressions

← Assignment  
(like = in Java)

= Equality testing  
(like == in Java)

*n*<sup>2</sup> Superscripts and other  
mathematical  
formatting allowed

# Main Point

For purposes of examining, analyzing, and comparing algorithms, a neutral algorithm language is used, independent of the particularities of programming languages, operating systems, and system hardware. Doing so makes it possible to study the inherent performance attributes of algorithms, which are present regardless of implementation details. This illustrates the SCI principle that more abstract levels of intelligence are more comprehensive and unifying.

# Overview of the Lesson

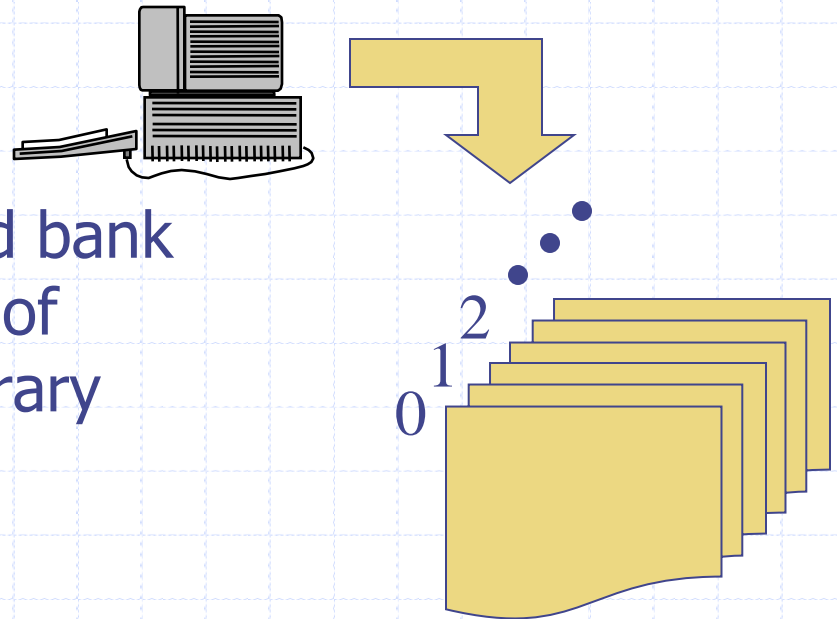
- Issues in Determining Running Time
- Pseudo-code to Describe Algorithms
- RAM Model and Counting Primitive Operations
- Asymptotic Algorithm Analysis
- Running Time of Recursive Algorithms
  - Guessing Method
  - Counting Self-Calls
  - The Master Formula

# The Random Access Machine (RAM) Model

◆ A **CPU**

◆ A potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or characters

◆ Memory cells are numbered and accessing any cell in memory takes unit time.



# Primitive Operations



- ◆ Basic computations performed by an algorithm
- ◆ Identifiable in pseudocode
- ◆ Largely independent of the programming language
- ◆ Assumed to take a constant amount of time in the RAM model



# Primitive Operations in This Course

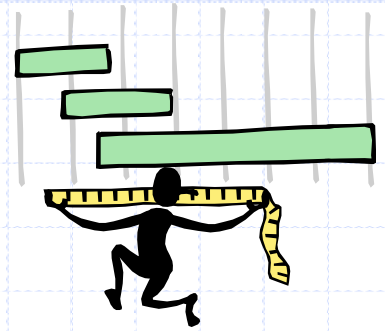
- Performing an arithmetic operation (+, \*, etc)
- Comparing two numbers
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method
- Following an object reference

# Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm <i>arrayMax</i> ( <i>A</i> , <i>n</i> )	# operations
<i>currentMax</i> $\leftarrow A[0]$	2
<i>m</i> $\leftarrow n - 1$	2
for <i>i</i> $\leftarrow 1$ to <i>m</i> do	$1 + n$
if <i>A</i> [ <i>i</i> ] > <i>currentMax</i> then	$2(n - 1)$
<i>currentMax</i> $\leftarrow A[i]$	$2(n - 1)$
{ increment counter <i>i</i> }	$2(n - 1)$
return <i>currentMax</i>	1
Total	$7n$

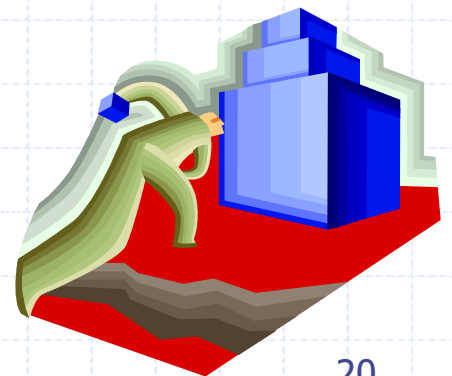
# Estimating Running Time



- ◆ Algorithm *arrayMax* executes  $7n$  primitive operations in the worst case. Define:
  - $a$  = Time taken by the fastest primitive operation
  - $b$  = Time taken by the slowest primitive operation
- ◆ Let  $T(n)$  be worst-case time of *arrayMax*. Then
$$a(7n) \leq T(n) \leq b(7n)$$
- ◆ Hence, the running time  $T(n)$  is bounded by two linear functions

# Growth Rate of Running Time

- ◆ Changing the hardware / software environment
  - Affects  $T(n)$  by a constant factor, but
  - Does not alter the growth rate of  $T(n)$
- ◆ The linear growth rate of the running time  $T(n)$  is an intrinsic property of algorithm *arrayMax*



# Overview of the Lesson

- Issues in Determining Running Time
- Pseudo-code to Describe Algorithms
- RAM Model and Counting Primitive Operations
- Asymptotic Algorithm Analysis
- Running Time of Recursive Algorithms
  - Guessing Method
  - Counting Self-Calls
  - The Master Formula

# Asymptotic Algorithm Analysis

- ◆ Asymptotic analysis of an algorithm looks at the growth rate of  $T(n)$  as  $n$  approaches infinity.
- ◆ Asymptotic analysis of an algorithm determines which complexity class the running time belongs to.
- ◆ To perform the (worst-case) asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function big-Oh notation (or one of its variants)
- ◆ Example:
  - We determined that algorithm *arrayMax* executes at most  $7n$  primitive operations
  - Since  $7n$  is  $O(n)$ , we say that algorithm *arrayMax* “runs in  $O(n)$  time”. Or we could say *arrayMax* runs in  $\Theta(n)$  time.

# Standard Complexity Classes

- ◆ The most common complexity classes used in analysis of algorithms are, in increasing order of growth rate:

$\Theta(1)$ ,  $\Theta(\log n)$ ,  $\Theta(n^{1/k})$ ,  $\Theta(n)$ ,  $\Theta(n \log n)$ ,  $\Theta(n^k)$  ( $k > 1$ ),  
 $\Theta(2^n)$ ,  $\Theta(n!)$ ,  $\Theta(n^n)$

Functions that belong to classes in the first row are known as *polynomial time bounded*.

# Basic Rules For Computing Asymptotic Running Times

## ◆ Rule-1: For Loops

The running time of a for loop is at most the running time of the statements inside the loop times the number of iterations (see *arrayMax*)

## ◆ Rule-2: Nested Loops

Analyze from inside out. The total running time of a statement inside a group of nested loops is the running time of the statement times the sizes of all the loops

```
for i ← 0 to n-1 do  
  for j ← 0 to n-1 do  
    k ← i + j
```

(Runs in  $\Theta(n^2)$  – or we can say runs in  $O(n^2)$  )



# (continued)

## ◆ Rule-3: Consecutive Statements

Running times of consecutive statements should be added in order to compute running time of the whole

```
for i ← 0 to n-1 do  
    a[i] ← 0  
for i ← 0 to n-1 do  
    for j ← 0 to i do  
        a[i] ← a[i] + i + j
```

(Running time is  $O(n) + O(n^2) = O(n^2)$  )

# (continued)

## ◆ Rule-4: If/Else

For the fragment

**if** *condition* **then**

S1

**else**

S2

the running time is never more than the running time of the *condition* plus the larger of the running times of S1 and S2.

# Overview of the Lesson

- Issues in Determining Running Time
- Pseudo-code to Describe Algorithms
- RAM Model and Counting Primitive Operations
- Asymptotic Algorithm Analysis
- Running Time of Recursive Algorithms
  - Guessing Method
  - Counting Self-Calls
  - The Master Formula

# Running Time of Recursive Algorithms: Guessing

**Problem:** Given an array of integers in sorted order, is it possible to perform a search for an element in such a way that no more than half the elements of the array are examined? (Assume the array has 8 or more elements.)

# Binary Search

**Algorithm** search(A,x)

*Input:* An already sorted array A with n elements and search value x

*Output:* true or false

**return** binSearch(A, x, 0, A.length-1)

**Algorithm** binSearch(A, x, lower, upper)

*Input:* Already sorted array A of size n, value x to be searched for in array section A[lower]..A[upper]

*Output:* true or false

**if** lower > upper **then return** false

mid  $\leftarrow$  (upper + lower)/2

**if** x = A[mid] **then return** true

**if** x < A[mid] **then**

**return** binSearch(A, x, lower, mid - 1)

**else**

**return** binSearch(A, x, mid + 1, upper)

# Example

Search key  $x = 7$

(1 2 5 7 12 **14** 21 24 25 38 52)

search left

(1 2 **5** 7 12)

search right

(**7** 12)

$A[\text{mid}] = 7 \Rightarrow \text{return true}$

# Example

Search key  $x = 20$

(1 2 5 7 **12** 14 21 24 25 38)

search right

(14 21 **24** 25 38)

search left

(**14** 21)

search right

(**21**)

search left

lower > upper  $\Rightarrow$  return false

# Binary Search Running Time

**Algorithm** binSearch(A, x, lower, upper)

*Input:* Already sorted array A of size n, value x to be searched for in array section A[lower]..A[upper]

*Output:* true or false

<b>if</b> lower > upper <b>then return</b> false	+1
mid $\leftarrow$ (upper + lower)/2	+3
<b>if</b> x = A[mid] <b>then return</b> true	+2
<b>if</b> x < A[mid] <b>then</b>	+2
<b>return</b> binSearch(A, x, lower, mid - 1)	
<b>else</b>	
<b>return</b> binSearch(A, x, mid + 1, upper)	+3 + T(n/2)

---

For the worst case (x is not in the array), running time is given by the **Recurrence Relation:**  
(In this case, right half is always exactly half the size of the original.)

$$T(1) = 13; \quad T(n) = 11 + T(n/2) \quad (\text{for all } n \geq 1)$$



# Solving A Recurrence Relation: The Guessing Method

$$T(1) = 13$$

$$T(2) = 11 + 13$$

$$T(4) = 11 + 11 + 13$$

$$T(8) = 11 + 11 + 11 + 13$$

$$T(2^m) = 11 * m + 13$$

$$T(n) = 11 * \log n + 13, \text{ which is } \Theta(\log n)$$

- ◆ Guessing method requires us to guess the general formula from a few small values.
- ◆ When using the guessing method, final formula should be verified. Sometimes induction is needed for this, though in simple cases, a direct verification is possible.

# Verification of Formula

**Claim** The function  $f(n) = 11 * \log n + 13$  is a solution to the recurrence

$$T(1) = 13; \quad T(n) = T(n/2) + 11 \quad (n \text{ a power of } 2)$$

**Proof:** Prove that whenever  $n$  is a power of 2,  $f(n)$  satisfies the recurrence. Since  $n$  is a power of 2, we write  $n = 2^m$ . Treating  $n$  as a power of 2,  $f(n)$  can be written as

$$f(2^m) = 11 * m + 13.$$

We must show that  $f(1) = 13$  and  $f(2^m) = f(2^{m-1}) + 11$

For  $n = 1$ , we have:

$$f(1) = 11 * \log 1 + 13 = 13$$

In general,

$$\begin{aligned} f(2^m) &= 11 * m + 13 \\ &= 11 * (m-1) + 11 + 13 \\ &= (11 * (m-1) + 13) + 11 \\ &= f(2^{m-1}) + 11, \end{aligned}$$

as required.

# Additional Points About the Guessing Method

- ◆ To state results (so far) correctly, it's necessary for  $n$  to be a power of 2. But we wish to have a bound on running time for any  $n$  – what can be done?

# Optional: Not a Power of 2 – Main Result

- ◆ A number-theoretic function  $f(n)$  is smooth if  $f(n)$  is eventually nondecreasing and if  $f(2n)$  is  $\Theta(f(n))$
- ◆ Theorem. Suppose  $f(n)$  is smooth and  $T(n)$  is eventually nondecreasing. Then if  $T(n)$  is  $\Theta(f(n))$  for  $n$  a power of 2,  $T(n)$  is  $\Theta(f(n))$ .

Note: All complexity functions  $f(n)$  we consider (except exponential functions) will be smooth, and running time functions  $T(n)$  will always be eventually nondecreasing. We will not verify this point in the future.

# The Divide and Conquer Algorithm Strategy

- ◆ The binary search algorithm is an example of a “Divide And Conquer” algorithm, which is typical strategy when recursion is used.
- ◆ The method:
  - **Divide** the problem into subproblems (divide input array into left and right halves)
  - **Conquer** the subproblems by solving them recursively (search recursively in whichever half could potentially contain target element)
  - **Combine** the solutions to the subproblems into a solution to the problem (return value found or indicate not found)

# Main Point

*Recurrence relations* are used to analyze recursively defined algorithms. Just as recursion involves repeated self-calls by an algorithm, so the complexity function  $T(n)$  is defined in terms of itself in a recurrence relation. Recursion is a reflection of the self-referral dynamics at the basis of Nature's functioning. Ultimately, there is just one field of existence; everything that happens therefore is just the dynamics of this one field interacting with itself. When individual awareness opens to this level of Nature's functioning, great accomplishments become possible.

# Overview of the Lesson

- Issues in Determining Running Time
- Pseudo-code to Describe Algorithms
- RAM Model and Counting Primitive Operations
- Asymptotic Algorithm Analysis
- Running Time of Recursive Algorithms
  - Guessing Method
  - Counting Self-Calls
  - The Master Formula

# Running Time of Recursive Algorithms: Counting Self-Calls

- ◆ To determine the running time of a recursive algorithm, another often-used technique is *counting self-calls*.
- ◆ Often, processing time in a recursion, apart from self-calls, is constant. In such cases, running time is proportional to the number of self-calls.



# Running time for Binary Search, Counting Self-Calls

- ◆ We can compute asymptotic running time of Binary Search using the technique of counting self-calls.
- ◆ Observation1: Suppose  $n$  is a power of 2 – say  $n = 2^m$ . Then the number of terms in the sequence  $2^m, 2^{m-1}, 2^{m-2}, \dots, 2^1, 2^0=1$  is  $m + 1 = 1 + \log n$ .
- ◆ Observation2: A more general fact that can be proved using this observation is if  $n$  is any positive integer, the sequence of terms  $n, n/2, n/4, \dots, n/2^m = 1$  where  $2^m$  is the largest power of 2 that is  $\leq n$ , has exactly  $1 + m = 1 + \lfloor \log n \rfloor$  terms
- ◆ In the worst case for BinarySearch, input size  $n$  is cut in half with each successive self-call. Therefore, the successive input sizes of self-calls is  $n/2, n/4, \dots, n/2^m = 1, 0$ , so  $1 + \lfloor \log n \rfloor$  self-calls are made. Running time is therefore  $\Theta(\log n)$ .

# Descending Sequences

Here are facts used in the last slide. We will revisit these in later lectures.

◆ If  $n$  is a power of 2, say  $n = 2^m$ , then the number of terms in the sequence

$$2^m, 2^{m-1}, 2^{m-2}, \dots, 2^1, 2^0 = 1$$

is  $m + 1 = 1 + \log n$ .

◆ A more general fact that can be proved using this observation is if  $n$  is any positive integer, the sequence of terms

$$n, n/2, n/4, \dots, n/2^m = 1$$

where  $2^m$  is the largest power of 2 that is  $\leq n$ , has exactly  $1 + m = 1 + \lfloor \log n \rfloor$  terms

# Overview of the Lesson

- Issues in Determining Running Time
- Pseudo-code to Describe Algorithms
- RAM Model and Counting Primitive Operations
- Asymptotic Algorithm Analysis
- Running Time of Recursive Algorithms
  - Guessing Method
  - Counting Self-Calls
  - The Master Formula

# The Master Formula

For recurrences that arise from Divide-and-Conquer algorithms (like Binary Search), there is a general formula for finding a closed-form solution:

**Theorem.** Suppose  $T(n)$  satisfies

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT(\lceil \frac{n}{b} \rceil) + cn^k & \text{otherwise} \end{cases}$$

where  $k$  is a non-negative integer and  $a, b, c, d$  are constants with  $a > 0, b > 1, c > 0, d \geq 0$ . Then

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^{\log_b a}) & \text{if } a > b^k \end{cases}$$

# Master Formula (continued)

## Notes.

- (1) The result holds if  $\lceil \frac{n}{b} \rceil$  is replaced by  $\lfloor \frac{n}{b} \rfloor$ .
- (2) Whenever  $T$  satisfies this “divide-and-conquer” recurrence, it can be shown that the conclusion of the theorem holds for *all* natural number inputs, not just to powers of  $b$ .

# Master Formula (continued)

**Example.** A particular divide and conquer algorithm has running time  $T$  that satisfies:

$$T(1) = d \quad (d > 0)$$

$$T(n) = 2T(n/3) + 2n$$

Find the asymptotic running time for  $T$ .

# Master Formula (continued)

**Solution.** The recurrence has the required form for the Master Formula to be applied. Here,

$$a = 2$$

$$b = 3$$

$$c = 2$$

$$k = 1$$

$$b^k = 3$$

Therefore, since  $a < b^k$ , we conclude by the Master Formula that

$$T(n) = \Theta(n).$$

# Master Formula - Exercise

- ◆ Use Master Formula to compute the running time of BinarySearch.



# Another example : The Fib Algorithm

- ◆ Next we will look at an algorithm that requires more advanced methods to determine the running time.
- ◆ To do this, we will make use of the techniques we have learned so far and develop them further.

# The Fib Algorithm

- ◆ The Fibonacci numbers are defined recursively by:  
 $F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$
- ◆ This is a recursive algorithm for computing the  $n$ th Fibonacci number:

**Algorithm** fib( $n$ )

**Input:** a natural number  $n$

**Output:**  $F(n)$

**if** ( $n = 0 \parallel n = 1$ ) **then return**  $n$

**return** fib( $n-1$ ) + fib( $n-2$ )

The running time of recursive Fib Algorithm is given by the following **Recurrence Relation:**

$$T(0) = b; T(1) = c; T(n) = T(n-1) + T(n-2) + d$$

( $b, c, d$  are some constants)

# The Fib Algorithm

◆ How to compute  $T(n)$ ?

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + d \\ &\geq T(n-2) + T(n-2) + d \\ &\geq 2T(n-2) \end{aligned}$$

**Lemma.** Suppose  $T(0) = b$ ,  $T(1) = c$ ,  $T(n) \geq 2T(n-2)$ . Define a recurrence  $S(0) = b$ ,  $S(1) = c$ ,  $S(n) = 2S(n-2)$ . Then for all  $n$ ,  
 $T(n) \geq S(n)$

**Proof.** Proceed by induction on  $n$  to show  $T(n) \geq S(n)$ . This is obvious for  $n = 0$  or  $1$ . Assume  $T(k) \geq S(k)$  whenever  $k < n$ . Then

$$T(n) \geq 2T(n-2) \geq 2S(n-2) = S(n)$$

In particular, if it can be shown that  $S(n)$  is  $\Theta(g(n))$ , then  $T(n)$  is  $\Omega(g(n))$ .

# Solving A Recurrence Relation: The Guessing Method

Question: How to compute  $S(n)$  given that  $S(1) = c$ ,  $S(n) = 2S(n-2)$ ?

$$S(1) = c$$

$$S(3) = 2 * S(1) = 2 * c$$

$$S(5) = 2 * S(3) = 2 * 2 * c = 2^2 c$$

$$S(7) = 2 * S(5) = 2 * 2 * 2 * c = 2^3 c$$

$$S(9) = 2 * S(7) = 2 * 2 * 2 * 2 * c = 2^4 c$$

$$S(n) = 2^{n/2} * c = (\sqrt{2})^n * c, \text{ which is } \Theta((\sqrt{2})^n)$$

Note: similarly, we can show  $S(n)$  is  $\Theta((\sqrt{2})^n)$  when  $n$  is even.

# Verification of Formula

**Claim** The function  $f(n) = 2^{n/2} * c$  is a solution to the recurrence  
 $S(1) = c, S(n) = 2S(n-2)$ . (when  $n$  is odd)

**Proof:**

Needs to show  $f(1) = c$  and  $f(n) = 2f(n-2)$

For  $n = 1$ , we have

$$f(1) = 2^0 * c = c$$

In general,

$$f(n) = 2^{n/2} * c = 2 * 2^{(n-2)/2} * c = 2f(n-2)$$

as required.

# The Fib Algorithm

By guessing method, we know that  $S(n)$  is  $\Theta((\sqrt{2})^n)$ ,  
therefore  $T(n)$  is  $\Omega((\sqrt{2})^n)$

This shows that fib is an *exponentially slow* algorithm!

An algorithm is said to have an *exponential running time* if its running time is  $\Theta(r^n)$  for some  $r > 1$ . We have shown here that fib is either exponential or worse! Fact: It can be shown there is a number  $\phi$  for which  $T(n)$  is  $\Theta(\phi^n)$  [ $\phi$  is called the *Golden Ratio*]

**DEMO:** See the code that accompanies this lecture.

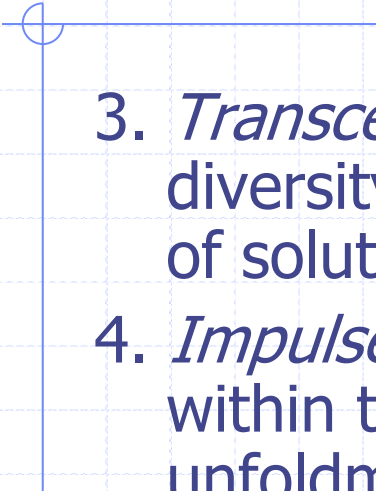
=====

**Fact:** Not only recursive fib algorithm runs exponentially, the Fibonacci numbers grow exponentially too. We will use this fact in later lecture.

**In-class Exercise:** For all  $n > 7$ ,  $F(n) > (\sqrt{2})^n$

# Connecting the Parts of Knowledge With The Wholeness of Knowledge

1. There are many techniques for analyzing the running time of a recursive algorithm. Most require special handling for special requirements (e.g.  $n$  not a power of 2, counting self-calls, verifying a guess).
2. The Master Formula combines all the intelligence required to handle analysis of a wide variety of recursive algorithms into a single simple formula, which, with minimal computation, produces the exact complexity class for algorithm at hand.

- 
3. *Transcendental Consciousness* is the field beyond diversity, beyond problems, and therefore is the field of solutions.
  4. *Impulses Within The Transcendental Field.* Impulses within this field naturally form the blueprint for unfoldment of the highly complex universe. This blueprint is called *Ved*.
  5. *Wholeness Moving Within Itself.* In Unity Consciousness, solutions to problems arise naturally as expressions of one's own unbounded nature.